

GO-05 02: Асинхронность. Goshedule, переменная окружения GOMAXPROCS

Описание:

Для понимания многопоточности в `golang` следует ознакомиться с планировщиком (`go scheduler`).

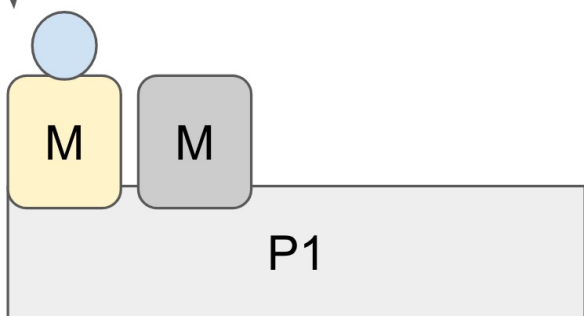
Обозначим основные сущности: `P` - процессор, ресурс, выполняющий код на языке Го. `M` - системный (рабочий) поток. `G` - горутина.

Горутина - легковесный поток - является абстракцией над системными потоками, требует меньше памяти для создания (2 КБ вместо 4 МБ). `G` распределяются по `M`. `M` распределяются по `P`. На каждом `P` может одновременно выполняться только один `M`. На каждом `M` может одновременно выполняться только одна `G`. Существует глобальная очередь горутин. Для каждого `P` существует локальная очередь `G` и очередь `M`.

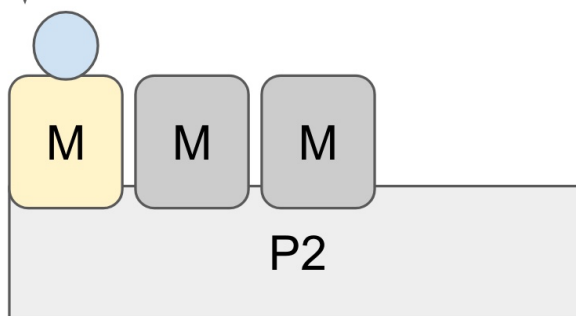
global queue



local queue



local queue

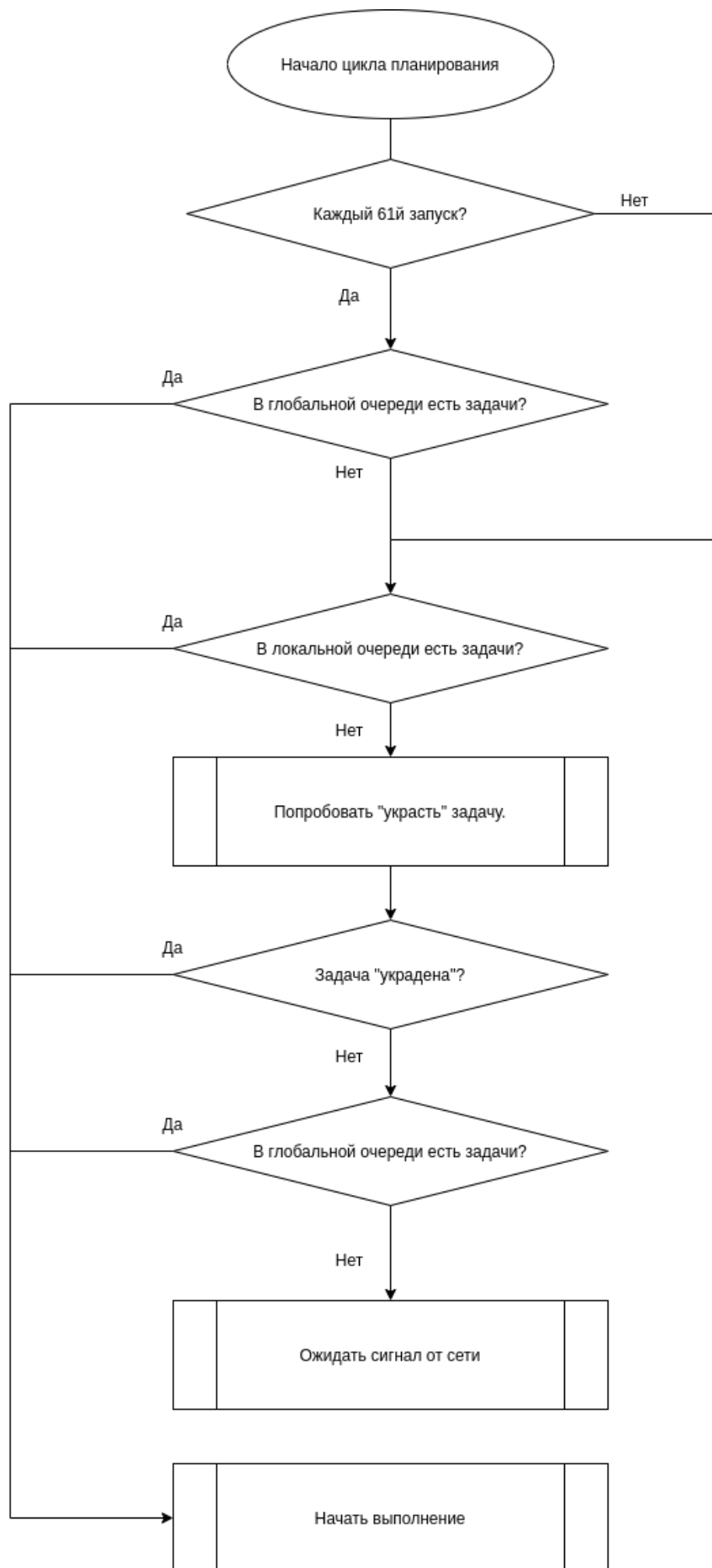


Планирование происходит циклично. Каждый цикл планирования заключается в поиске горутин, которая готова к тому, чтобы быть запущенной, и ее исполнении.

Поиск происходит в следующем порядке:

- каждый 61й запуск проверить глобальную очередь `G`;
- если не найдено, проверить локальную очередь;
- если не найдено, то:
 - попытаться «украсть» у других `P`;

- если не вышло, проверить глобальную очередь;
- если все равно не вышло, поллить (poll) сеть.



Кража (stealing) - принцип работы планировщика, при котором незанятый P пытается забрать себе половину локальной очереди G у другого P. Для сравнения, существует (не в go, а вообще) принцип «поделиться» (sharing) - при котором занятый P пытался бы распределить свои задачи между незанятыми.

Переменная GOMAXPROCS отвечает за максимально допустимое количество используемых программой P.

До версии go 1.14 GOMAXPROCS=1 мог существенно повлиять на работу программы.

Например, такой код

```
package main
import (
    "fmt"
    "time"
)

func main() {
    go func() {
        var u int
        for {
            if u == 1 {
                break
            }
        }
    }()
    <-time.After(time.Millisecond * 10)

    fmt.Println("go version is > 1.14")
}
```

не выдал бы сообщение go version is > 1.14 и не закончил бы выполнение.

Пример запуска:

```
GOMAXPROCS=1 go run main.go
```

В версии go 1.14 планировщик претерпел некоторые изменения, которые позволили улучшить эффективность его работы и избежать ситуации, подобные описанной выше. Но почему до 1.14 поток выполнения уходил в бесконечный цикл? Все дело в том, что go scheduler мог запланировать вытеснение горутин из процесса выполнения только в следующих случаях:

- вызов функции (при определенных условиях);
- блокировки (мьютексы, каналы);
- системные вызовы (syscalls);

- `runtime.Gosched()`.

Собственно, и получается, что в ситуации выше ни одного из этих событий не наступало и горутина навсегда занимала единственный поток выполнения (`GOMAXPROCS=1`).

В 1.14 к этому списку добавилось асинхронное вытеснение, которое и решило вышеописанную проблему.

При желании, можно ознакомиться с исходниками по ссылке

<https://golang.org/src/runtime/proc.go> или попробовать отследить работу планировщика с помощью `trace` <https://golang.org/cmd/trace/> (это выходит за рамки программы, предлагается для самостоятельного ознакомления).

Полезные ссылки:

- [src/runtime/proc.go](https://golang.org/src/runtime/proc.go)
- [Work-stealing планировщик в Go](#)
- [Планировщик Go](#)
- [GO Scheduler: теперь не кооперативный?](#)

Задание:

1. Создайте в проекте `module05` ветку `module05_02`.
2. Используя код из предыдущего задания, с помощью `GOMAXPROCS` добейтесь поочередного выполнения функций подсчета 32го и 33го чисел Фибоначчи, запущенных в отдельных горутинах, и вывода их в консоль, отображая при этом спиннер. Устанавливайте `GOMAXPROCS` путем добавления строки `runtime.GOMAXPROCS(1)` в начало функции `main`. Учтите, что для вытеснения основного потока нужно вызвать какую-то из блокирующих операций (например, `time.Sleep()`).
3. В ответе пришлите ссылку на MR в ветку `master` своего проекта ветки `module05_02`.